

```

import numpy as np
import random
import matplotlib.pyplot as plt

# Define the new objective function: f(x) = x * cos(x) - sin(2x)
def objective_function(x):
    return x * np.cos(x) - np.sin(2 * x)

# Particle Swarm Optimization (PSO) implementation
class Particle:
    def __init__(self, dimension, lower_bound, upper_bound):
        # Initialize the particle position and velocity randomly
        self.position = np.random.uniform(lower_bound, upper_bound, dimension)
        self.velocity = np.random.uniform(-1, 1, dimension)
        self.best_position = np.copy(self.position)
        self.best_value = objective_function(self.position[0]) # Evaluate first dimension

    def update_velocity(self, global_best_position, w, c1, c2):
        # Update the velocity of the particle
        r1 = np.random.rand(len(self.position))
        r2 = np.random.rand(len(self.position))

        # Inertia term
        inertia = w * self.velocity

        # Cognitive term (individual best)
        cognitive = c1 * r1 * (self.best_position - self.position)

        # Social term (global best)
        social = c2 * r2 * (global_best_position - self.position)

        # Update velocity
        self.velocity = inertia + cognitive + social

    def update_position(self, lower_bound, upper_bound):
        # Update the position of the particle
        self.position = self.position + self.velocity

        # Ensure the particle stays within the bounds
        self.position = np.clip(self.position, lower_bound, upper_bound)

    def evaluate(self):
        # Evaluate the fitness of the particle based on the objective function
        fitness = objective_function(self.position[0]) # Using only the first dimension for this 1D problem

        # Update the particle's best position if necessary
        if fitness < self.best_value:
            self.best_value = fitness
            self.best_position = np.copy(self.position)

def particle_swarm_optimization(dim, lower_bound, upper_bound, num_particles=30, max_iter=100, w=0.5, c1=1.5, c2=1.5):
    # Initialize particles
    particles = [Particle(dim, lower_bound, upper_bound) for _ in range(num_particles)]

    # Initialize the global best position and value
    global_best_position = particles[0].best_position
    global_best_value = particles[0].best_value

    # To track progress (fitness values over time and positions)
    fitness_history = []
    positions_history = []

    for i in range(max_iter):
        # Update each particle
        for particle in particles:
            particle.update_velocity(global_best_position, w, c1, c2)
            particle.update_position(lower_bound, upper_bound)
            particle.evaluate()

        # Update global best position if needed
        if particle.best_value < global_best_value:
            global_best_value = particle.best_value
            global_best_position = np.copy(particle.best_position)

    # Record the best fitness and particle positions for this iteration
    fitness_history.append(global_best_value)

```

```

    positions_history.append([particle.position[0] for particle in particles]) # Store positions of particles (1D)

    # Optionally print the progress
    if (i + 1) % 10 == 0:
        print(f"Iteration {i + 1}/{max_iter} - Best Fitness: {global_best_value}")

    return global_best_position, global_best_value, fitness_history, positions_history

# Plotting function for PSO progress
def plot_pso_progress(fitness_history, positions_history, global_best_position, lower_bound, upper_bound, max_iter):
    # Plot Fitness Over Time
    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)
    plt.plot(range(1, max_iter + 1), fitness_history, color='blue', label='Global Best Fitness')
    plt.title('Fitness Convergence Over Time')
    plt.xlabel('Iterations')
    plt.ylabel('Best Fitness Value')
    plt.grid(True)

    # Plot Particle Positions Over Time
    plt.subplot(1, 2, 2)
    for i in range(max_iter):
        plt.scatter(positions_history[i], [i] * len(positions_history[i]), color='blue', alpha=0.5, label=f'Iteration {i+1}' if i == 0 else
        plt.scatter(global_best_position, max_iter, color='red', marker='X', label='Global Best Position')
    plt.title('Particle Positions Over Iterations')
    plt.xlabel('Position')
    plt.ylabel('Iteration')
    plt.grid(True)

    plt.tight_layout()
    plt.show()

# Set the parameters for the PSO algorithm
dim = 1 # One-dimensional problem
lower_bound = -10 # Lower bound of the search space
upper_bound = 10 # Upper bound of the search space
num_particles = 30 # Number of particles in the swarm
max_iter = 100 # Number of iterations

# Run the PSO
best_position, best_value, fitness_history, positions_history = particle_swarm_optimization(
    dim, lower_bound, upper_bound, num_particles, max_iter
)

# Output the best solution found
print("\nBest Solution Found:")
print("Position:", best_position)
print("Fitness:", best_value)

# Plot the progress of the PSO
plot_pso_progress(fitness_history, positions_history, best_position, lower_bound, upper_bound, max_iter)

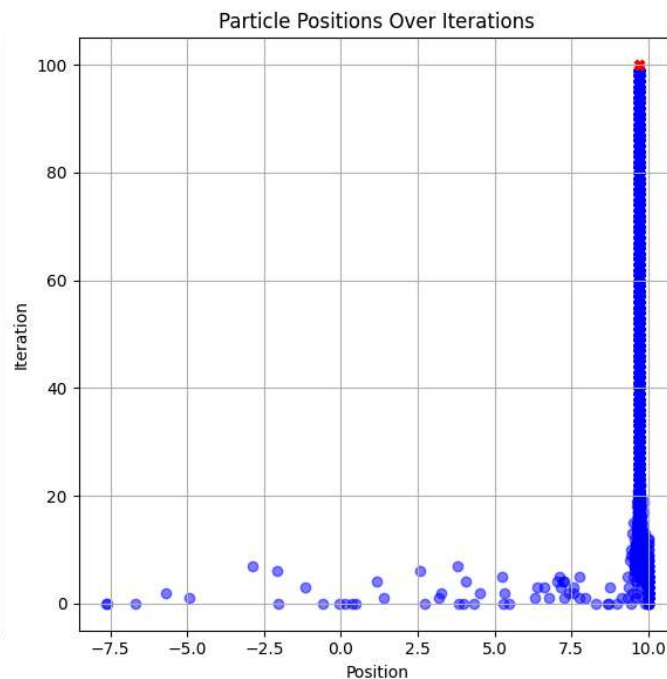
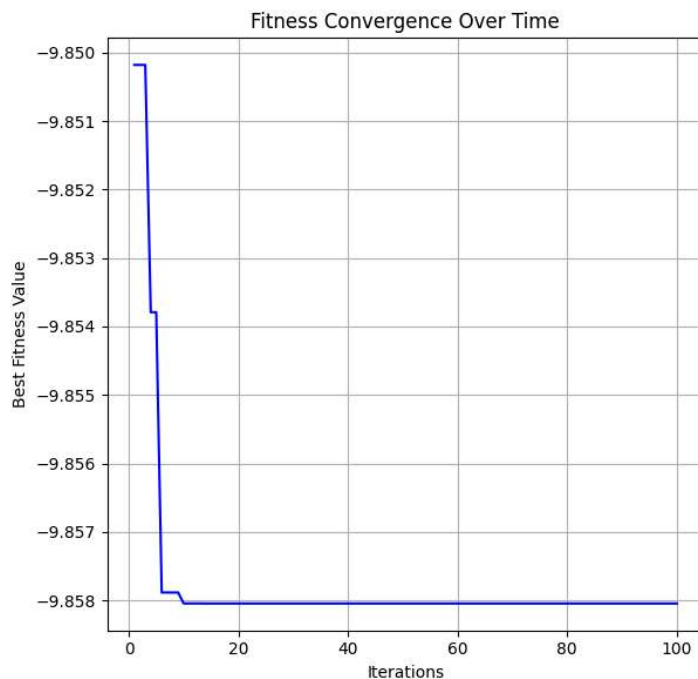
```

```

Iteration 10/100 - Best Fitness: -9.8580439672337
Iteration 20/100 - Best Fitness: -9.85804488649004
Iteration 30/100 - Best Fitness: -9.858044886603542
Iteration 40/100 - Best Fitness: -9.858044886759474
Iteration 50/100 - Best Fitness: -9.858044886759485
Iteration 60/100 - Best Fitness: -9.858044886759487
Iteration 70/100 - Best Fitness: -9.858044886759487
Iteration 80/100 - Best Fitness: -9.858044886759487
Iteration 90/100 - Best Fitness: -9.858044886759487
Iteration 100/100 - Best Fitness: -9.858044886759487

```

Best Solution Found:  
Position: [9.70257695]  
Fitness: -9.858044886759487



```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random

# Step 1: Define the objective function for portfolio optimization (Sharpe ratio)
def objective_function(weights, mean_returns, cov_matrix, risk_free_rate=0.02):
    """
    Objective function to calculate the negative Sharpe ratio for the portfolio.
    :param weights: Portfolio weights (array of length N)
    :param mean_returns: Expected returns of the assets (array of length N)
    :param cov_matrix: Covariance matrix of asset returns
    :param risk_free_rate: Risk-free rate (default 2%)
    :return: Negative Sharpe ratio (since PSO minimizes the objective function)
    """
    weights = np.array(weights)

    # Ensure no short-selling (weights between 0 and 1)
    weights = np.clip(weights, 0, 1)

    # Normalize the weights so they sum to 1, with a check to avoid division by zero
    weight_sum = np.sum(weights)
    if weight_sum == 0: # If the sum of weights is zero, set them to be equal
        weights = np.ones_like(weights) / len(weights) # Equal allocation
    else:
        weights = weights / weight_sum # Normalize the weights

    # Calculate expected portfolio return and portfolio variance
    expected_return = np.dot(weights, mean_returns)
    portfolio_variance = np.dot(weights.T, np.dot(cov_matrix, weights))
    portfolio_volatility = np.sqrt(portfolio_variance)

```

```

# If volatility is 0 (no risk), we return a very large number (i.e., a very poor portfolio)
if portfolio_volatility == 0:
    return np.inf

# Calculate the Sharpe ratio: (return - risk-free rate) / volatility
sharpe_ratio = (expected_return - risk_free_rate) / portfolio_volatility
return -sharpe_ratio # Minimize the negative Sharpe ratio

# Step 2: Define the Particle class to represent a particle in the PSO algorithm
class Particle:
    def __init__(self, dimension, lower_bound, upper_bound):
        """
        Initialize the particle with random position and velocity.
        :param dimension: Number of assets (dimension of the portfolio)
        :param lower_bound: Lower bound for asset allocation
        :param upper_bound: Upper bound for asset allocation
        """
        self.position = np.random.uniform(lower_bound, upper_bound, dimension) # Random position (weights)
        self.velocity = np.random.uniform(-1, 1, dimension) # Random velocity
        self.best_position = np.copy(self.position) # Best known position for this particle
        self.best_value = np.inf # Best value (objective function value, initially very high)

    def update_velocity(self, global_best_position, w, c1, c2):
        """
        Update particle velocity based on PSO update rule.
        :param global_best_position: Global best known position (optimal portfolio weights)
        :param w: Inertia weight
        :param c1: Cognitive coefficient
        :param c2: Social coefficient
        """
        r1 = np.random.rand(len(self.position)) # Random number for cognitive component
        r2 = np.random.rand(len(self.position)) # Random number for social component

        # Inertia: keeps the particle moving in its previous direction
        inertia = w * self.velocity

        # Cognitive component: pushes the particle towards its personal best position
        cognitive = c1 * r1 * (self.best_position - self.position)

        # Social component: pushes the particle towards the global best position
        social = c2 * r2 * (global_best_position - self.position)

        # Update velocity
        self.velocity = inertia + cognitive + social

    def update_position(self, lower_bound, upper_bound):
        """
        Update the particle's position (weights) based on its velocity.
        :param lower_bound: Lower bound for asset weights
        :param upper_bound: Upper bound for asset weights
        """
        self.position = self.position + self.velocity

        # Ensure the particle's position is within bounds (weights between 0 and 1)
        self.position = np.clip(self.position, lower_bound, upper_bound)

    def evaluate(self, mean_returns, cov_matrix, risk_free_rate):
        """
        Evaluate the fitness of the particle using the objective function.
        :param mean_returns: Expected returns of the assets
        :param cov_matrix: Covariance matrix of asset returns
        :param risk_free_rate: Risk-free rate for Sharpe ratio calculation
        """
        fitness = objective_function(self.position, mean_returns, cov_matrix, risk_free_rate)

        # Update the particle's best position if necessary
        if fitness < self.best_value:
            self.best_value = fitness
            self.best_position = np.copy(self.position)

# Step 3: Define the PSO algorithm for portfolio optimization
def particle_swarm_optimization(mean_returns, cov_matrix, num_particles=30, max_iter=100, w=0.7, c1=1.5, c2=1.5):
    """
    Main PSO function for portfolio optimization.
    :param mean_returns: Expected returns of the assets
    :param cov_matrix: Covariance matrix of asset returns
    :param num_particles: Number of particles in the swarm

```

```

:param max_iter: Number of iterations for the optimization
:param w: Inertia weight
:param c1: Cognitive coefficient
:param c2: Social coefficient
:return: Best portfolio weights, best Sharpe ratio, and fitness history
"""

# Number of assets in the portfolio (dimension of the problem)
dim = len(mean_returns)

# Initialize particles
particles = [Particle(dim, 0, 1) for _ in range(num_particles)]

# Initialize the global best position and value
global_best_position = particles[0].best_position
global_best_value = particles[0].best_value

fitness_history = []

# Main optimization loop
for i in range(max_iter):
    # Update each particle's velocity, position, and evaluate fitness
    for particle in particles:
        particle.update_velocity(global_best_position, w, c1, c2)
        particle.update_position(0, 1)
        particle.evaluate(mean_returns, cov_matrix, risk_free_rate=0.02)

    # Update global best if this particle has a better fitness value
    if particle.best_value < global_best_value:
        global_best_value = particle.best_value
        global_best_position = np.copy(particle.best_position)

    # Record the best Sharpe ratio at this iteration
    fitness_history.append(-global_best_value) # The objective function is negative Sharpe ratio

    # Optionally, print progress every 10 iterations
    if (i + 1) % 10 == 0:
        print(f"Iteration {i + 1}/{max_iter} - Best Sharpe Ratio: {-global_best_value}")

return global_best_position, -global_best_value, fitness_history

# Step 4: Load the stock data and calculate returns and covariance matrix
def load_stock_data(tickers, start_date, end_date):
    """
    Load stock data from Yahoo Finance and calculate returns and covariance matrix.
    :param tickers: List of stock symbols (e.g., ['AAPL', 'MSFT', 'GOOGL'])
    :param start_date: Start date for historical data (format: 'YYYY-MM-DD')
    :param end_date: End date for historical data (format: 'YYYY-MM-DD')
    :return: mean_returns (average returns) and cov_matrix (covariance matrix)
    """
    import yfinance as yf

    # Download stock data
    data = yf.download(tickers, start=start_date, end=end_date)['Adj Close']

    # Calculate daily returns
    returns = data.pct_change().dropna()

    # Calculate mean returns and covariance matrix
    mean_returns = returns.mean()
    cov_matrix = returns.cov()

    return mean_returns, cov_matrix

# Step 5: Run the PSO optimization on stock data
tickers = ['AAPL', 'MSFT', 'GOOGL', 'AMZN', 'TSLA'] # Example tickers
start_date = '2020-01-01'
end_date = '2023-01-01'

# Load stock data and compute returns and covariance matrix
mean_returns, cov_matrix = load_stock_data(tickers, start_date, end_date)

# Run PSO to find optimal portfolio
best_position, best_sharpe, fitness_history = particle_swarm_optimization(mean_returns, cov_matrix)

# Step 6: Output the best portfolio weights and Sharpe ratio
print("\nBest Portfolio Weights:")
print(best_position)

```

```
print("Best Sharpe Ratio:", best_sharpe)
```

```
↳ [*****100%*****] 5 of 5 completed
Iteration 10/100 - Best Sharpe Ratio: -0.37461750264850174
Iteration 20/100 - Best Sharpe Ratio: -0.37461750264850174
Iteration 30/100 - Best Sharpe Ratio: -0.37461750264850174
Iteration 40/100 - Best Sharpe Ratio: -0.37461750264850174
Iteration 50/100 - Best Sharpe Ratio: -0.37461750264850174
Iteration 60/100 - Best Sharpe Ratio: -0.37461750264850174
Iteration 70/100 - Best Sharpe Ratio: -0.37461750264850174
Iteration 80/100 - Best Sharpe Ratio: -0.37461750264850174
Iteration 90/100 - Best Sharpe Ratio: -0.37461750264850174
Iteration 100/100 - Best Sharpe Ratio: -0.37461750264850174

Best Portfolio Weights:
[0. 0. 0. 0. 1.]
```